Université de Lille

TP 5.05
JPA

R5.A.09 - Frameworks WEB
BUT3 - Info
Philippe Mathieu
2024–2025

Objectifs

Prise en main de JPA. Comprendre les annotations propres à JPA et la manière de gérer les objets.

Exercice 1 : Les bases de JPA - avec une seule table

Q1. Créez un projet Maven `tp505Exo1` pour une application JAVA classique **(pas de web)** avec l'archetype `maven-archetype-quickstart`, un groupID `fr.but3` et comme nom de projet `tp505Exo1`

```
mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart -DgroupId=fr.but3 -DartifactId=tp505Exo1
```

Q2. Complétez l'arborescence en ajoutant le répertoire `resources` dans `src/main`.

```
mkdir -p tp505Exo1/src/main/resources
```

Q3. Corrigez le `pom`, en ajoutant les propriétés classiques ainsi que les dépendances `postgres`, `eclipselink` et `jpa` (Ne rien mettre concernant le WEB : ni cargo, ni maven-war-plugin).

```
<properties>
  <maven.compiler.source>17</maven.compiler.source>
  <maven.compiler.target>17</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <exec.mainClass>${project.groupId}.App</exec.mainClass>
</properties>

<dependencies>
<!-- JPA -->
<dependency>
 <groupId>org.eclipse.persistence</groupId>
 <artifactId>eclipselink</artifactId>
 <version>4.0.3</version>
</dependency>
<!-- EclipseLink -->
<dependency>
 <groupId>org.eclipse.persistence</groupId>
 <artifactId>org.eclipse.persistence.jpa</artifactId>
 <version>4.0.3</version>
</dependency>
<!-- postgresql -->
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.7.4</version>
</dependency>
```

Q4. Testez immédiatement `mvn clean package exec:java` . A priori aucune erreur ne doit être générée et `Hello World` doit s'afficher.

Q5. Créez un répertoire `model` dans `fr/but3` ainsi qu'une classe `Auteur.java` avec `ano`, `nom`, `prenom`, `email` comme attributs privés, les getters, les setters et un `toString`. Inutile d'ajouter un constructeur, JPA se contente du constructeur vide présent par défaut. Ajoutez à cet objet les annotation `@Entity` et `@Id`

Q6. Dans `resources/META-INF`, configurez le `persistence.xml` afin qu'il gère cet objet en mode `drop-and-cr` Si nécessaire un `persistence.xml` minimal est disponible sur Moodle.

Q7. Modifiez le `persistence.xml` pour qu'un fichier `data.sql` soit chargé par défaut. Augmentez le niveau de trace de JPA à `FINE`. Créez le fichier `data.sql` et mettez quelques données dedans (A Dumas, V Hugo, E Zola, A Christie ...).

Vous devriez avoir maintenant l'arborescence suivante :

```
--- src
     |--- main
            |--- java
            |      |--- fr
            |             |--- but3
            |                    |--- App.java
            |                    |--- model
            |                           |--- Auteur.java
            |--- resources
                      |--- META-INF
                      |          |--- persistence.xml
                      |--- data.sql
```

Q8. Créez l'objet entity manager dans `App` puis relancez le projet. Dans la trace de lancement vous devriez maintenant voir les ordres SQL exécutés (y compris ceux du `data.sql`)

Q9. Regardez la structure et le contenu de la table `Auteur` créée dans votre base.

Q10. Modifiez le `main` afin de retrouver l'un des objets insérés via sa clé, puis affichez le.

Q11. Modifiez le `persistence.xml` en ajoutant les propriétés de génération de scripts SQL de création et destruction des tables. Après chaque exécution assurez vous d'avoir **pour information**, un fichier `creer.sql` et un fichier `detruire.sql` qui indiquent les ordres que JPA exécute en interne pour le DDL.

Q12. Observez le comportement (et le type d'erreur) :

- si le nom du *persistance unit* ne correspond pas entre le fichier xml et votre programme.
- si `persistence.xml` est renommé `persistance.xml`
- si le nom de la base de données est incorrect

On constate que JPA s'est occupé de créer la table, qu'il recrée cette table à chaque lancement, qu'il remplit les données et qu'il est capable de faire le lien objet-relationnel.

Exercice 2 : Gestion des numéros automatiques et autres contraintes

Q1. Faites une copie totale du précédent projet que vous renommez en `tp505Exo2`

Q2. Modifiez le type de l'attribut `ano` pour qu'il soit géré par un numéro automatique avec l'annotation `GenerationType.IDENTITY`

Q3. Modifiez la colonne `nom` en ajoutant des contraintes de longueur (annotation `@column`)

Q4. Modifiez en conséquence votre fichier de données

Q5. Regardez dans la trace et dans les scripts obtenus ce qui est généré lors de la création de la table

Exercice 3 : Utilisation avancée - plusieurs tables

Q1. Recopiez le projet précédent en `tp505Exo3`

Q2. Créez un objet `Livre(lno,categorie,titre)`

Pour l'instant ces deux objets sont indépendants. Dans la suite de cet exercice nous étendons le modèle de 3 manières différentes :

- Lien mono-directionnel : accéder à l'auteur par le livre
- Lien mono-directionnel : accéder aux livres par l'auteur
- Lien bi-directionnel

A chacune des étapes suivantes regardez attentivement les tables générées, et le nom de la clé étrangère (voir slides 32 à 34). Un moyen simple et rapide consiste à taper :

```
mvn clean package exec:java ; cat creer.sql
```

Q3. Etape 1. On souhaite d'abord pouvoir accéder à l'auteur à partir d'un livre. Ajoutez la référence à l'objet `Auteur` dans la classe `Livre` avec une annotation `@ManyToOne`)

Q3.1. Quel est le nom de la clé étrangère générée ?

Q3.2. Pour renommer cette clé, il est possible d'utiliser l'annotation `@JoinColumn`. Renommez la en `ano` (histoire d'avoir le même nom que dans la table Auteur) et vérifiez.

Q3.3. Ecrire le code qui permet de retrouver un livre (par son id) et afficher son auteur (modifiez le toString de Livre). Pensez à ajouter quelques livres à vos auteurs dans `data.sql`

```
Livre livre = em.find(Livre.class, 1);
System.out.println(livre);
```

Etape 2. On souhaite maintenant mettre en place le lien inverse. Pour cela mettre dans `Livre` ce qui concerne `Auteur` en commentaire.

Q4. On souhaite donc pouvoir accéder aux livres à partir d'un auteur. Ajoutez dans `Auteur` une collection de livres, (annotation `@OneToMany`)

Q4.1. Regardez les tables générées, vous constatez que comme précédemment la clé étrangère a été générée correctement

Q4.2. Ecrire le code qui permet de rechercher un `Auteur` (par son id) et afficher ses livres

Q5. Etape 3. On souhaite maintenant mettre en place un lien bi-directionnel. Il faut donc un attribut `Auteur` dans la classe `Livre` et une collection de `Livre` dans la classe `Auteur`. Indiquez un `MappedBy` pour éviter une table supplémentaire. Vérifiez que tout fonctionne correctement.

Exercice 4 : Passage au WEB

Q1. Passer à une application WEB. Il y a 2 manières de passer à une application WEB. Soit ajouter `webapp` dans la hiérarchie et mettre le packaging à war, soit recréer toute une arborescence avec le `maven-archetype-weba`
Nous utiliserons cette fois la première solution.

Q2. Ajoutez au pom votre serveur web embarqué `cargo`

Q3. L'application web nécessite un `webapp/WEB-INF/web.xml`. Il est possible de l'éviter en ajoutant au POM la propriété

```
<failOnMissingWebXml>false</failOnMissingWebXml>
```

Q4. Dans `webapp`, ajouter une page jsp `index.jsp` avec un `Hello World` dedans.

Q5. Testez si `http://localhost:8080/tp505Exo3` fonctionne correctement.

Q6. Créez une JSP `lister.java` qui permet de lister chaque auteur de la base dans un `H1` avec tous ses livres dans un `ol`.

Exercice 5 : La cerise sur le gâteau

Q1. Lombok est une librairie Java qui s'intègre automatiquement à votre éditeur et à vos outils de construction, pour vous éviter d'écrire le code répétitif des POJO (constructeurs, getters, setters, toString etc ...).

Q1.1. Installez Lombok pour votre éditeur. Pour VSCODE c'est une extension à ajouter, Pour Eclipse un plugin à installer.

Q1.2. Ajoutez la dépendance Lombok à votre pom

Q1.3. Modifiez le code des POJO en supprimant tout le code qu'il contient et en le remplaçant par des annotations Lombok. par exemple :

```
@Data
@AllArgsConstructor
@NoArgsConstructor
```

Q2. Le SGBD H2 est un petit couteau-suisse qui peut fonctionner en mode TCP, en mode embedded et en mode `inMemory`. Ce dernier va nous permettre d'embarquer directement la base de données dans le projet, sans plus avoir besoin de BDD externe.

Q2.1. Ajoutez la dépendance H2

Q2.2. Modifiez votre `persistence.xml` afin d'utiliser H2 en mode `inMemory`. Dans ce mode de fonctionnement, l'url doit être de la forme `jdbc:h2:mem:lapin;MODE=LEGACY`

Q2.3. Retestez votre application et vérifiez que tout fonctionne correctement.